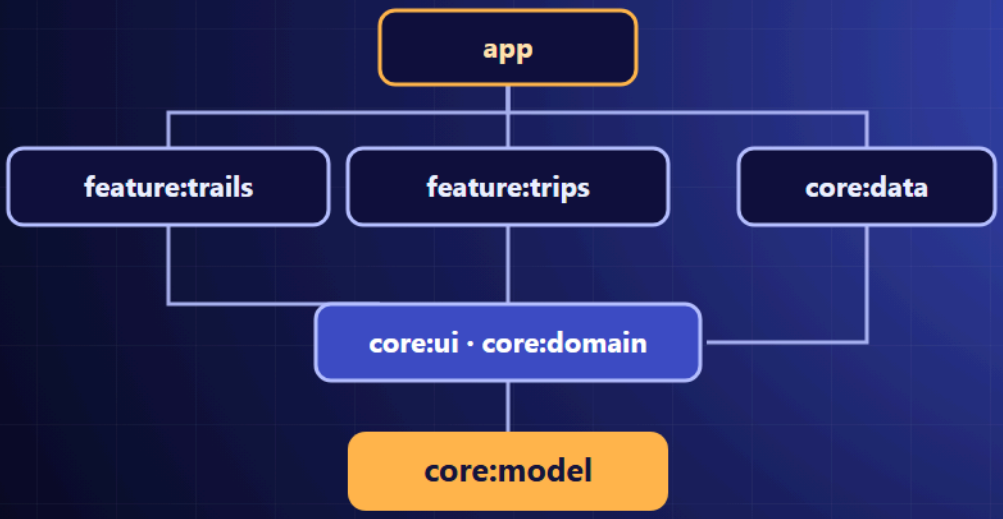
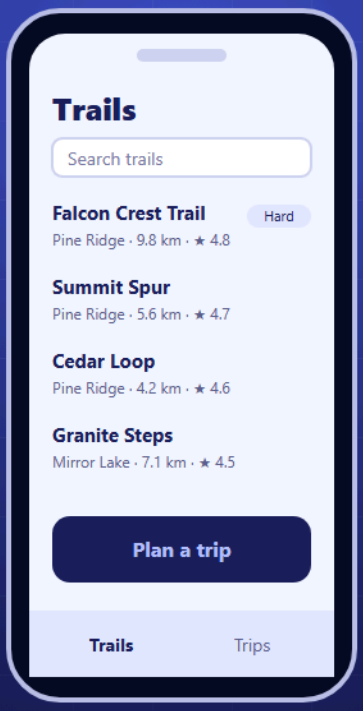





```
@HiltViewModel
class TrailListViewModel @Inject
constructor(searchTrails: SearchTrailsUseCase)
  : ViewModel() {
  val uiState: StateFlow<TrailListUiState>
    = results.stateIn(viewModelScope, …)
}
```

```
@Module
@Binds TrailRepository
@Provides Clock
@AssistedFactory
```

A PRACTICAL GUIDE FOR GROWING APPS

Android App Architecture

MVVM and MVI, clean layers, modularization, and dependency injection with Hilt

UiState · StateFlow · Use Cases · Offline-First Repositories
Hilt · Convention Plugins · Feature Modules · Testing · ADRs

STRUCTURE · TESTABILITY · MODULES · TEAMS

THENEXBOOKS

Android App Architecture

MVVM and MVI, Clean Layers, Modularization, and Dependency Injection with Hilt for Apps That Scale From a single tangled module to a modular, testable Kotlin app a whole team can grow

First Edition

THENEXBOOKS

Android App Architecture
MVVM and MVI, Clean Layers, Modularization, and Dependency Injection with Hilt for Apps That Scale

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Android, Kotlin, Gradle, Jetpack libraries, and Android Studio change from release to release; plugin versions, menu names, compiler messages, and recommended practices can differ between versions. Always consult the current official documentation for the versions you use.

Android, Google Play, Jetpack Compose, and Android Studio are trademarks of Google LLC. Kotlin is a trademark of the Kotlin Foundation. Gradle is a trademark of Gradle, Inc. Other product and library names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Google LLC, JetBrains, the Kotlin Foundation, or Gradle, Inc. The cover artwork, diagrams, and interface sketches are original and do not reproduce any trademarked logo or artwork.

Waypoint, the hiking trip planner used as the running example in this book, is fictional, as are Northbeam Labs, its team members, users, trails, regions, trips, web addresses, and all sample data. Domains use the reserved `.example` top-level domain; no real server is needed to follow the book.

Preface

Most Android apps do not fail because one screen is hard to build. They fail slowly. The first release ships quickly, and then every feature takes a little longer than the one before. A change to the trail list breaks trip planning. Nobody wants to touch the singleton that "does the data". Tests are rare because nothing can be created without the whole app behind it. Two developers cannot work on two features without stepping on each other, the build gets slower every month, and a new team member needs weeks to learn where anything lives.

Architecture is the set of decisions that keeps that from happening, or that lets you climb back out once it has. It is not a library you add to `build.gradle.kts` or a diagram you draw once. It is a handful of ideas applied consistently: state that flows in one direction, a single source of truth for every piece of data, explicit dependencies, a domain layer that knows nothing about screens or databases, and modules that make the right thing easy and the wrong thing impossible to compile.

This book teaches those ideas by working on one app the way a real team would. We start from Google's recommended layered architecture, model screens with ViewModels and immutable UI state, build the same kind of feature in MVVM and in MVI and compare them honestly, give business rules a home in use cases, make the data layer offline-first, wire everything with Hilt, split the build into core and feature modules with convention plugins, and finish with the practices that keep a codebase healthy for years: dependency rules enforced by the build, incremental migration of legacy code, and architecture decision records. At every step we explain the trade-off: what a pattern costs and when you should not use it.

Who This Book Is For

- **Android developers** who are comfortable with Kotlin and Jetpack Compose and want apps that stay pleasant to change as they grow.
- **Tech leads and senior developers** who need shared conventions, a module structure several people can work in at once, and a vocabulary for architecture decisions.
- **Developers maintaining an older codebase** with singletons, callbacks, and one giant module who want a realistic, step-by-step path to a modern architecture without a rewrite.
- **Developers coming from other platforms** who know architecture concepts and want to see how they map to ViewModel, Kotlin flows, Hilt, Room, and Gradle.

We assume you can build a Compose screen, know Kotlin's classes, lambdas, and null safety, and have used coroutines and a ViewModel at a basic level. We do not re-teach the basics; we focus on structure.

What You Will Learn

- Google's recommended app architecture: the UI, domain, and data layers, what belongs in each, and where it is reasonable to deviate.

- Unidirectional data flow, a single source of truth, and how state travels from a Room table to a composable and back.
- How to model screen state with ViewModel and immutable UI state, and how to expose it safely with `StateFlow`, `stateIn`, and lifecycle-aware collection.
- MVVM and MVI in practice, side by side on real Waypoint screens, with a clear way to choose between them.
- How to handle one-off events, user messages, and process death without losing data or showing a message twice.
- How to map network DTOs, database entities, domain models, and UI models, and where each mapper belongs.
- How to build offline-first repositories and background synchronization, use cases that earn their place, and error handling that crosses layers cleanly.
- Dependency injection from first principles, then Hilt in depth: components, modules, bindings, scopes, qualifiers, multibindings, assisted injection, and replacing bindings in tests.
- How to modularize: core and feature modules, convention plugins in `build-logic`, navigation between features that do not know each other, and dependency rules the build enforces.
- How to test every layer on the JVM, migrate legacy code incrementally, record decisions with ADRs, and keep an architecture healthy as a team grows.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains why architecture matters, maps Google's recommended layers onto a real app, and covers unidirectional data flow, ViewModel and UI state, and the reactive plumbing of flows, lifecycles, and scopes. **Part II — Presentation Patterns: MVVM and MVI** (Chapters 6–10) builds the Trails feature in MVVM and the Plan Trip screen in MVI, compares the two, handles one-off events and process death, and maps models between layers. **Part III — Domain and Data Layers** (Chapters 11–15) covers repositories, offline-first writes and synchronization, use cases, error handling, and designing for testability. **Part IV — Dependency Injection with Hilt** (Chapters 16–20) moves from manual dependency injection to Hilt's components, scopes, qualifiers, multibindings, assisted injection, and testing. **Part V — Modularization** (Chapters 21–24) explains core and feature modules, convention plugins, navigation between feature modules, and enforcing module boundaries. **Part VI — Scaling a Real App** (Chapters 25–28) migrates a legacy feature incrementally, records decisions in ADRs, looks at team-scale practices, and closes with an end-to-end case study. The appendices provide a cheat sheet, templates and reference code, a production architecture checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Waypoint

Throughout the book we work on **Waypoint**, a fictional hiking trip planner made by **Northbeam Labs**, a small product studio. Waypoint lets you browse and search trails, open a trail's details, plan a trip with a date and party size, and see your upcoming and finished trips. Its sample data is small enough to check by hand: eight trails in three regions (Pine Ridge, Mirror Lake, and Fernhollow) and three trips for our sample user, Alex Rivera. When Alex plans a hike on Falcon Crest Trail for June 22, 2026, you can follow that single action through every layer of the app, from a button tap to a row in the database and back to the screen.

Waypoint already has history. Its first version, **Waypoint Classic**, was a single-module app built around singletons and callbacks. Most of it has been rebuilt, but one legacy island remains, and in Part VI we migrate it without stopping releases. By the end of the book, Waypoint is a modular app with a thin app module that acts as its composition root, a pure Kotlin domain module, an offline-first data layer over Room and a REST API, feature modules that never depend on each other, Hilt wiring throughout, convention plugins that keep every build file a few lines long, and fast JVM tests at every layer.

Waypoint's backend is fictional too. The app ships with an in-memory demo data source, and every test uses fakes or a local mock server, so you never need a real server, an account, or a network connection to follow along.

Versions and Conventions

At the time of writing, the code uses **Kotlin 2.4**, **Android Gradle Plugin 9.4** with its built-in Kotlin support, **Gradle 9.8**, **Jetpack Compose** through the 2026.09 bill of materials, **Navigation Compose 2.10** with type-safe routes, **Hilt 2.60**, and **Room 2.8**, with a minimum SDK of 26 and a target SDK of 36. All builds and tests in the book were run with JDK 21. Android tooling changes quickly; plugin versions, menu names, and compiler messages may differ slightly in the version you use, so when something is not exactly as described, check the current documentation. Third-party libraries are discussed as options only; Waypoint's code uses Jetpack, Kotlin, and a small set of widely used open-source libraries.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause bugs, data loss, or maintenance problems.

Code appears in `monospace`, with a caption above each listing that shows the file it belongs to, such as `feature/trips/.../plan/PlanTripViewModel.kt`; the dots stand for the package folders `src/main/kotlin/com/northbeam/waypoint/feature/trips`. Long lines are wrapped to fit the page. Terminal commands are shown as `./gradlew` for macOS and Linux; on Windows use `.\gradlew.bat`. Test output is trimmed to the lines that matter. Results that need an emulator or a device are introduced with "you should see something like", because exact output varies between devices and tool versions. Diagrams and screen sketches are drawn in text so they stay readable in any PDF viewer.

Contents

Preface

PART I — FOUNDATIONS

1 Why Architecture Matters: Goals, Trade-offs, and Meet Waypoint

2 Google's Recommended Architecture: UI, Domain, and Data Layers

3 Unidirectional Data Flow and a Single Source of Truth

4 ViewModel and UiState: Modeling Screen State

5 Flows, Lifecycles, and Scopes: The Reactive Plumbing

PART II — PRESENTATION PATTERNS: MVVM AND MVI

6 MVVM in Practice: Building the Trails Feature

7 MVI in Practice: Intents, Reducers, and Side Effects

8 MVVM vs MVI: Choosing, Mixing, and Avoiding Cargo Cults

9 One-Off Events, User Messages, and Process Death

10 Mapping Models Between Layers: DTOs, Entities, Domain Models, and UI Models

PART III — DOMAIN AND DATA LAYERS

11 Repositories and the Single Source of Truth in the Data Layer

12 Offline-First Writes and Synchronization

13 The Domain Layer: Use Cases That Earn Their Place

14 Error Handling Across Layers

15 Designing a Testable Architecture

PART IV — DEPENDENCY INJECTION WITH HILT

16 Dependency Injection Fundamentals: Manual DI and the Composition Root

17 Hilt Essentials: Components, Modules, and Bindings

18 Scopes, Qualifiers, and Multibindings

19 Assisted Injection and Runtime Arguments

20 Testing with Hilt: Replacing Bindings on the JVM

PART V — MODULARIZATION

21 Why and How to Modularize: Core and Feature Modules

22 Convention Plugins and build-logic: Taming Gradle Configuration

23 Navigation Between Feature Modules

24 Module Boundaries, API Surfaces, and Dependency Rules

PART VI — SCALING A REAL APP

25 Migrating Legacy Code Incrementally

26 Architecture Decision Records for Waypoint

27 Scaling the Team and the Codebase

28 Case Study: Adding Waypoint Journal End to End

APPENDICES

A Architecture Cheat Sheet

B Templates and Reference Code

C Production Architecture Checklist

D Glossary

PART I

Foundations

Why architecture matters and what it costs, Google's recommended UI, domain, and data layers mapped onto a real app, state that flows in one direction from a single source of truth, ViewModels with immutable UI state, and the flows, lifecycles, and scopes that connect them.

1 · Why Architecture Matters

2 · Google's Recommended Architecture

3 · Unidirectional Data Flow

4 · ViewModel and UiState

5 · Flows, Lifecycles, and Scopes

CHAPTER 1 · PART I — FOUNDATIONS

Why Architecture Matters: Goals, Trade-offs, and Meet Waypoint

Architecture is the set of decisions in an app that are expensive to change later. In this chapter we define what that means for an Android app, name the goals good structure serves and the costs it always brings, and meet Waypoint, the hiking app whose codebase we will grow, test, and reshape for the rest of the book.

What "Architecture" Means for an Android App

Ask ten Android developers what architecture is and you will hear ten answers: "MVVM", "Clean Architecture", "Hilt", "modules", "whatever the lead says". Each of these is a *tool* or a *pattern*. None of them is the thing itself. The definition we use throughout this book is deliberately plain:

Architecture is the set of decisions that are expensive to change once code depends on them.

Picking a color for a button is not an architectural decision; you change one line and move on. Deciding where the list of trails lives, who is allowed to modify it, and how a screen learns that it changed *is* architectural, because dozens of classes will be written on top of that answer. If you get it wrong, every one of those classes has to change with it.

This definition has two useful consequences. First, it tells us where to spend our care: on the boundaries that many pieces of code lean on, not on the inside of a single function. Second, it reminds us that architecture is always about the *future*. A structure that looks over-engineered for the code you have today can be exactly right for the code you will have in six months, and a structure that feels lean today can turn into a trap. Good architecture is a bet about which changes are likely, made explicit so the team can check it.

On Android, that bet has some platform-specific inputs. Your process can be killed at any moment while the app is in the background. Activities are destroyed and recreated when the user rotates the device or changes the system language. The network is often slow or absent. The UI toolkit, Jetpack Compose, redraws the screen from state, so the shape of that state matters more than it did in the View era. A good Android architecture is one that makes these facts boring: the app keeps working through them without heroics in every screen.

Patterns are vocabulary, not goals

Because the word "architecture" is so overloaded, teams sometimes adopt a pattern for its own sake. Someone reads that use cases are good, so every repository call gets wrapped in a use case that does nothing. Someone reads that modules are good, so a ten-screen app gets forty modules. In this book we treat patterns as shared vocabulary: names for solutions that already exist, so that "this screen is MVI" or "this is a pass-through use case" means the same thing to everyone on the team. A pattern earns its place only when it solves a problem you can name in your own codebase. When we introduce one, we will always say what it costs and when we would *not* use it.

What Good Structure Buys You

If architecture is a bet, what are we betting on? Across the Waypoint codebase we optimize for five goals. They are not ranked; real decisions trade them against each other.

1. Change

The single most important property of a codebase is how cheaply it absorbs change. Product requirements move: Priya, our product designer, will ask for a new sort order, a new filter, a new screen. The backend moves: Sam will version the API. Platform rules move: new Android releases tighten background limits and permissions. A well-structured app localizes each kind of change. A new sort order should touch the search rule and the screen, not the database. A new API version should touch the network layer, not every ViewModel.

2. Testability

Code that is easy to test is code whose dependencies are visible and replaceable. If a ViewModel creates its own Retrofit client, you cannot test it without a server. If it receives a `TrailRepository` interface, you can hand it a fake in one line. Testability is not a separate goal bolted on at the end; it is the most reliable early signal that your boundaries are in the right place. When a test is painful to write, the design is usually telling you something.

3. Parallel work

A codebase that one person can hold in their head does not need much structure. The moment two or three people work on it at once, they start to collide: merge conflicts in the same giant file, one person's refactor breaking another's feature, nobody sure who owns which class. Clear layers and modules give people separate places to work and clear contracts between those places.

4. Onboarding

New developers learn a codebase by pattern matching. If every screen follows the same shape (a route, a stateful screen, a stateless content composable, a ViewModel exposing one state object), a newcomer who understands one screen understands all of them. Consistency is an architectural feature, and it is one of the cheapest ones to have.

5. Working offline

Waypoint is used on trails, which is where cell coverage goes to die. The app must show saved trails and let people plan trips without a connection, then catch up later. Offline support is not a feature you can sprinkle on at the end; it decides where the truth about your data lives. That is why we treat it as an architectural goal rather than a feature request.

What Structure Costs

Every one of those benefits has a price, and pretending otherwise is how teams end up with ceremony nobody can justify. We will name the costs explicitly whenever we add structure.

- **Indirection.** An interface between the ViewModel and the database means one more file to open when you trace a bug. A use case between them means another. Each layer makes local reasoning easier and global navigation harder.
- **Ceremony.** Separate models for the network, the database, the domain, and the UI mean mapper functions that copy fields from one to the next. For small apps, this can be more code than the features themselves.
- **Build complexity.** Modules need Gradle files, dependency rules, and convention plugins. A dependency injection framework adds annotation processing and its own error messages. These are real learning costs for every new team member.
- **Premature commitment.** Structure encodes assumptions. If you split the app into modules along the wrong seams, moving code between them later is harder than moving it within one module.

The right amount of architecture is therefore not "as much as possible". It is the least structure that keeps the expected changes cheap. A weekend prototype needs almost none. An app with several developers, offline data, and a multi-year roadmap needs a lot. Most apps live somewhere in between and move along that line over time.

> **WARNING**
>
> Be suspicious of any claim, including one you find in a blog post or hear at a conference, that a pattern makes apps "faster", "scalable", or "maintainable" without saying what it costs and in which situation. Architecture advice that has no trade-offs is marketing. We will try hard not to give you any.

Signs Your App Needs More Structure

How do you know when to invest? We find it more useful to look for symptoms than to follow a checklist of patterns. Here are the ones we see most often:

- **Fear of touching a file.** One Activity or ViewModel has grown so large that everyone avoids changing it, and when they must, they copy code instead.

- **Bugs that move.** You fix a stale list on one screen and it appears on another, because two screens keep their own copies of the same data.
- **Tests that need the world.** Testing a simple rule ("a free user may have three open trips") requires an emulator, a database, and a network stub.
- **Rotation bugs.** Data disappears or a request fires twice when the device rotates, because state lives in the wrong object.
- **Merge-conflict hot spots.** The same handful of files appear in nearly every pull request.
- **Slow onboarding.** New teammates need weeks before they can ship a small change confidently, because every screen works differently.

None of these symptoms is solved by a single pattern. Each points at a missing boundary: between the UI and its data, between screens and shared state, between rules and frameworks. Much of this book is about putting those boundaries in the right places.

Meet Northbeam Labs and Waypoint

Talking about architecture in the abstract only goes so far. Every chapter in this book works on the same codebase: **Waypoint**, a hiking trip planner and tracker built by **Northbeam Labs**, a small fictional product studio. The team has four people, and you will see them make, argue about, and record decisions:

Person	Role	Owns
Lena Ortiz	Android lead	The `core:*` modules and the build
Sam Okafor	Backend	The Waypoint API
Priya Natarajan	Product and design	UX copy and screen flows
Theo Brandt	QA	The test strategy

The app's package and application ID is `com.northbeam.waypoint`. Its backend lives at `https://api.waypoint.example/v1/`, which, as the `.example` domain suggests, does not exist. The app ships with an in-memory data source that serves eight sample trails, and the tests use fakes or a local mock server. You will never need a network connection or an account to follow along.

What Waypoint does

The full Waypoint product has six feature areas. Each book in the thenexbooks Android series builds the slice of the app its topic needs, so this list is larger than what we build here:

1. **Trails:** browse and search trails, cached locally, with a detail screen showing distance, elevation, and rating.
2. **Trips:** plan a trip on a trail (date, party size, checklist), record a hike, and see a summary of distance, duration, and elevation gain.

3. **Journal:** private notes and photos per trip.
4. **Sync:** offline-first storage; trips and journal entries are written locally and synced when online.
5. **Waypoint Plus:** an optional subscription that unlocks unlimited trips and offline trail packs.
6. **Notifications:** trip reminders and sync failures.

This book builds the *architecture* of Waypoint rather than every pixel of it. Concretely, we work on the Trails feature (written in MVVM style), the Plan Trip screen (written in MVI style), the My Trips list, the offline-first data layer with background sync, the dependency injection graph built with Hilt, and the modular Gradle build that holds it all together. In Part VI we migrate a piece of legacy code, the old trip checklist, and we finish by adding the Journal feature end to end as a case study that uses everything from the earlier chapters.

The sample data you will see everywhere

Waypoint's data is fixed so that every example, test, and output in the book is reproducible. There are eight trails, with IDs `tr-001` to `tr-008`:

ID	Name	Region	km / gain	Difficulty, rating
tr-001	Cedar Loop	Pine Ridge	4.2 / 120 m	EASY, 4.6
tr-002	Falcon Crest Trail	Pine Ridge	9.8 / 640 m	HARD, 4.8
tr-003	Lakeshore Path	Mirror Lake	6.5 / 80 m	EASY, 4.3
tr-004	Granite Steps	Mirror Lake	7.1 / 455 m	MODERATE, 4.5
tr-005	Old Mill Ridge	Fernhollow	11.3 / 720 m	HARD, 4.1
tr-006	Meadow Run	Fernhollow	3.4 / 45 m	EASY, 4.0
tr-007	Summit Spur	Pine Ridge	5.6 / 510 m	MODERATE, 4.7
tr-008	Brookside Way	Mirror Lake	8.0 / 210 m	MODERATE, 4.4

The signed-in user is always **Alex Rivera** (`u-100`), on the free plan. Sign-in itself is out of scope for this book. Alex already has three trips: Cedar Loop on Saturday, June 20, 2026 for two people, Granite Steps on Saturday, July 4, 2026 for four, and a finished hike on Lakeshore Path on May 30, 2026. In every test, and in every example where the date matters, "today" is **Monday, June 15, 2026, at 08:00 UTC**. The code never calls `LocalDate.now()` directly; it asks an injected `Clock`, and the tests pass a fixed one. That single decision, which we justify properly in Part III, is what makes date rules testable.

Waypoint Classic: A Cautionary Tale

Waypoint was not born with the structure you are about to see. Version 1.x, which the team now calls **Waypoint Classic**, shipped from 2019 to 2023 as a single Gradle module built with Android Views. It worked, it had happy users, and it was very hard to change.

Its design was the one many apps arrive at by accident. Each screen was an Activity that did everything: it inflated the layout, made network calls, read and wrote `SharedPreferences`, and formatted strings. Shared data lived in singletons such as a `TrailCache` object and a `ChecklistManager` object that any screen could reach and mutate. There were no interfaces to replace, so there were almost no unit tests; the team tested by hand before each release.

The symptoms from the previous section all showed up. The trail list and the trail detail screen each kept their own copy of the data and disagreed after a refresh. Rotating the device during a refresh fired a second request. Adding the "free accounts can plan three open trips" rule meant changing four Activities, because the check had been copied wherever trips were created.

None of this was anyone's fault. A single-module app with singletons is a perfectly sensible way to find out whether a product has users. The problem was that the structure never changed as the product and team grew. In 2025 the team rewrote Trails and Trips with Jetpack Compose and split the codebase into modules, a decision recorded as ADR-0007 (Architecture Decision Records are the subject of Chapter 26). The trip checklist was left behind as a legacy island; we migrate it in Chapter 25, and you will see first-hand how to modernize code like this without a risky big-bang rewrite.

Every other scar from Waypoint Classic has a chapter that removes it. Chapter 3 gives trails a single source of truth, so the list and the detail screen can no longer disagree, and shows why rotation no longer fires a second request. Chapter 13 moves the open-trip limit into one use case instead of four Activities. Chapter 15 builds the automated test suite the old app never had, Chapter 16 retires the singletons in favor of injected dependencies, and Chapter 21 retells the split from one module into many.

> **NOTE**
>
> The repository you download with this book already contains the post-2025 structure. When a chapter explains *why* a module or layer exists, it is explaining a decision the team already made. Chapter 21 retells the move from one module to many, step by step, so you can apply the same sequence to your own app.

A Tour of the Repository

The companion starter project is a complete, building Gradle project named `waypoint`. Here is its layout, with one line on what each part is for:

Repository layout

```
waypoint/
  settings.gradle.kts          includeBuild("build-logic"); 11 modules
  build.gradle.kts             plugins (apply false) + shared test settings
  gradle.properties            parallel, build cache, configuration cache on
  gradle/libs.versions.toml    version catalog, incl. waypoint-* plugin aliases
  build-logic/convention/      convention plugins (Chapter 22)
  app/                         composition root: application, activity, NavHost
```

```
core/model/                JVM: Trail, Trip, TrackPoint, JournalEntry, User
core/common/               Android: dispatchers, app scope, Clock bindings
core/domain/               JVM: repository interfaces, results, use cases
core/data/                 Android: offline-first repositories, mappers
core/database/             Android: Room database, entities, DAOs
core/network/              Android: network data source, Retrofit, demo data
core/ui/                   Android + Compose: theme, shared UI, messages
core/testing/              JVM: TestData, fake repositories, test rules
feature/trails/            Trails list (MVVM) and trail detail
feature/trips/             Plan Trip (MVI) and My Trips
```

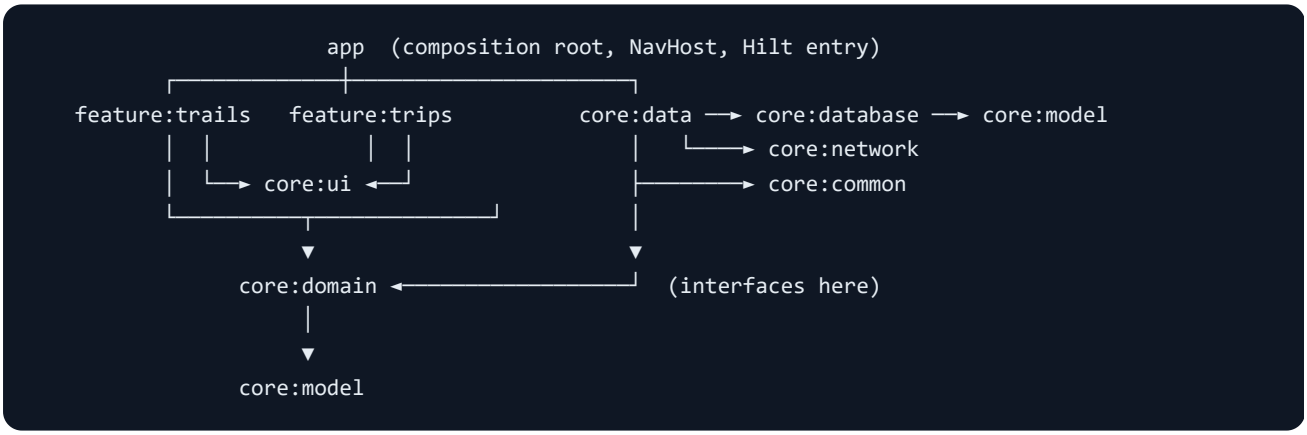
Notice the naming scheme. `core:*` modules hold code shared by features: models, rules, data access, common UI. `feature:*` modules hold one user-facing feature each. The `app` module holds almost no logic at all. It is the **composition root**, the one place that knows every module and wires them together into a running app. Packages follow the module path: `com.northbeam.waypoint.core.domain`, `com.northbeam.waypoint.feature.trails`, and so on, with the app itself at `com.northbeam.waypoint`.

Two of the core modules, `core:model` and `core:domain`, are plain Kotlin/JVM modules with no Android dependency at all. That is a deliberate constraint. It means the models and the business rules compile and test on a plain JVM in seconds, and it makes it physically impossible for a rule to reach for a `Context`.

The module graph

Modules are only useful if the dependencies between them follow rules. This is Waypoint's graph; arrows point from a module to what it depends on:

Waypoint module graph



Repository interfaces live in `core:domain`; `core:data` implements them. Three rules fall out of this picture, and we will enforce them in code in Chapter 24:

- **Features never depend on other features.** The Trails feature does not know the Trips feature exists. When tapping "Plan a trip" on a trail moves the user into the Trips feature, the `app` module wires that navigation.
- **Features never depend on `core:data`, `core:database`, or `core:network`.** They talk to repository *interfaces* in `core:domain`. Only `app` knows which implementation is used.

- **`core:domain` and `core:model` never depend on Android.**

The middle rule is the most unusual one, and it is worth a preview. In Google's default picture of the layers, the domain layer depends on the data layer. Waypoint inverts that: the repository interfaces live in `core:domain`, and `core:data` implements them. Chapter 2 explains both options and why the team chose this one (ADR-0002); for now, just notice that the arrow from `core:data` points *into* `core:domain`.

What a feature module looks like

Because the shared Gradle configuration lives in convention plugins, a feature module's entire build file is tiny. This is the real build file of the Trails feature:

`feature/trails/build.gradle.kts`

```
plugins {
    alias(libs.plugins.waypoint.android.feature)
}

android {
    namespace = "com.northbeam.waypoint.feature.trails"
}
```

The `waypoint.android.feature` plugin applies the Android library and Compose setup, Hilt, Kotlin serialization for navigation routes, and the allowed dependencies on `core:model`, `core:domain`, and `core:ui`. Adding a new feature module therefore takes a few lines rather than a page of copied configuration. Chapter 22 builds these plugins from scratch.

Where the patterns live

You will meet each of these in detail later, but it helps to know where to look:

Concept	Where in Waypoint	Chapter
Screen state (MVVM)	`TrailListViewModel`, `TrailListUiState`	4, 6
Screen state (MVI)	`PlanTripViewModel`, `PlanTripReducer`	7
Business rules	`SearchTrailsUseCase`, `PlanTripUseCase`	13
Single source of truth	`OfflineFirstTrailRepository` + Room	3, 11
Errors across layers	`DataResult`, `DataError`	14
Dependency injection	Hilt modules in `core:*` and `app`	16–20
Build structure	`build-logic/convention`	21–24

Building the Starter Project

Before reading further, build the project once. Doing so proves your toolchain works and gives you a green baseline: every later change you make can be checked against it.

What you need

- **JDK 21** to run Gradle. The project compiles to Java 17 bytecode, but the build itself needs a recent JDK. Android Studio bundles a suitable one in recent versions; on the command line, point `JAVA_HOME` at a JDK 21 installation.
- **The Android SDK** with the platform matching `compileSdk` installed. Android Studio's SDK Manager can install it; a `local.properties` file with `sdk.dir` tells Gradle where it lives.
- **Android Studio**, in a recent version that supports the Android Gradle Plugin version listed below. You can also work entirely from the command line; every command in this book works there.

You do *not* need an emulator or a device for most of the book. The tests run on the JVM: plain unit tests for the domain, Robolectric for Room, Compose UI, and Hilt. When a chapter shows something that needs a device, such as instrumented tests, we say so.

Run the build

From the repository root, run the unit tests and build a debug APK:

Terminal

```
# macOS / Linux
./gradlew test assembleDebug

# Windows
.\gradlew.bat test assembleDebug
```

The first run downloads the Gradle distribution and every dependency, so it takes a while. When it finishes you should see the familiar ending:

Build output (excerpt)

```
> Task :core:model:compileKotlin
> Task :core:domain:test
…
> Task :core:data:testDebugUnitTest
> Task :feature:trails:testDebugUnitTest
> Task :feature:trips:testDebugUnitTest
…
> Task :app:testDebugUnitTest
> Task :app:packageDebug
> Task :app:assembleDebug
```

```
BUILD SUCCESSFUL in …
Configuration cache entry stored.
```

The debug APK lands in `app/build/outputs/apk/debug/app-debug.apk`, and each module's test reports are under its own `build/reports/tests/` folder. The starter's test suite covers the domain rules, the Room DAOs, the network and data layers, the ViewModels, a Compose UI test, and a Hilt graph test, all on the JVM. The second run is much faster, because the build cache and the configuration cache (both switched on in `gradle.properties`) let Gradle skip work whose inputs have not changed.

> **TIP**
> On Windows, always type `.\gradlew.bat` with the leading `.\`. Some shells are configured not to run programs from the current directory, and the bare `gradlew.bat` then fails with a "not recognized" error even though the file is right there.

If the build fails, the three most common causes are a JDK older than 21 running Gradle (check with `./gradlew --version`, which prints the JVM it uses), a missing SDK platform (the error names it), or an Android Studio version too old for the Android Gradle Plugin. The last one shows up as a sync error in the IDE even though the command-line build works; updating Android Studio fixes it.

Target versions

At the time of writing, the starter uses these versions. They are pinned in `gradle/libs.versions.toml`, so you can see and change them in one place:

Tool or library	Version	Tool or library	Version
Kotlin	2.4.20	Hilt / Dagger	2.60.1
Android Gradle Plugin	9.4.1	Room	2.8.5
Gradle	9.8.0	Navigation Compose	2.10.2
KSP	2.3.12	Lifecycle	2.11.0
Compose BOM	2026.09.00	Coroutines	1.11.0
compileSdk / targetSdk	37 / 36	WorkManager	2.12.0
minSdk	26	Retrofit / OkHttp	3.0.0 / 5.5.0

> **NOTE**
> Libraries move quickly. The architectural ideas in this book do not depend on exact versions, but code listings are verified against the versions above. If a newer version renames or deprecates an API we use, the idea still holds; check the library's release notes for the new spelling.